Allison (Ruthie) Houston, Evey Wright, Jacky Lin, Kiara Johnson

CS 335

Batts

2/10/2026

Project 1 Report

In our project, we improved the functionality across multiple files, including RayTracer.cpp, light.cpp, material.cpp, cubeMap.cpp, and trimesh.cpp to implement the Whitted illumination model, ray-tracing, Phong interpolation, and cube mapping.

RayTracer.cpp

In RayTracer.cpp, the trace() function receives inputs x and y and shoots the primary ray through a pixel at coordinates x and y. It calls traceRay() to compute the color of the pixel. TracePixel is then used to convert pixel indices into normalized coordinates. These functions are used throughout RayTracer.cpp, first to check if the ray intersects anything in the scene and store the intersection distance if there is an intersection. Then, the program computes local Phong shading along with reflection if recursion depth allows. If there is no intersection, cube mapping is enabled and the cube map is sampled. Otherwise, a sky gradient background is returned. The following code snippet shows the if-statement used to make recursion stop when a maximum depth is reached or when all RGB components fall below the threshold, thus avoiding unnecessary recursive calls:

```
if (depth <= 0 ||
 (thresh[0] < this->thresh &&
 thresh[1] < this->thresh &&
 thresh[2] < this->thresh)) {
```

Additionally, we used the following if-statement to check for a cube map and return the correct color from it when a ray doesn't intersect with any objects:

```
if (traceUI->cubeMap()) {
  return traceUI->getCubeMap()->getColor(r);
}
```

light.cpp

In light.cpp, lighting is calculated and implemented into the ray tracer. The program defines both DirectionalLight and PointLight, which are each defined by distance attenuation, direction, and shadow calculation. DirectionalLight always has a distance attenuation of 1 because directional lights are infinitely far and do not dim with distance, while PointLight uses the quadratic falloff formula 1 / (1 + 0.1 * d^2) because point light dims with distance. The following code snippet shows our implementation of shadow attenuation for point lights :

```
double d = glm::length(position - P);

 // Quadratic attenuation: 1 / (1 + c * d^2)
 double atten = 1.0 / (1.0 + 0.1 * d * d);

 // Clamp so we never brighten the light
 return std::min(1.0, atten);
```

The direction of DirectionalLight is constant for all points, while PointLight calculates the direction from each point towards the light's position. Finally, for DirectionLight, shadow calculation checks to see if the point is blocked by any objects by casting a shadow ray opposite of the light's direction. If any object intersects that ray, the point is fully shadowed; otherwise, it is lit. For PointLight, a shadow is shot toward the point light. The following code snippet shows our implementation of shadow attenuation for point lights:

```
glm::dvec3 PointLight::shadowAttenuation(const ray &, const glm::dvec3 &p) const {
  glm::dvec3 toLight = position - p;
  double maxDist = glm::length(toLight);
  glm::dvec3 L = glm::normalize(toLight);
  const double eps = 1e-6;
  ray shadowRay(p + eps * L, L, glm::dvec3(1.0, 1.0, 1.0), ray::SHADOW);
  isect i;
  if (scene->intersect(shadowRay, i)) {
    if (i.getT() < maxDist) {
      return glm::dvec3(0.0, 0.0, 0.0); // blocked before light
    }
  }
  return glm::dvec3(1.0, 1.0, 1.0);
}
```

The shadow calculation also checks if a blockade is actually between the point and the light. If something blocks the ray before reaching the light distance, the point is in shadow. If not, the point is fully lit.

material.cpp

The file material.cpp contains a shade function, which implements the Whitted illumination model. It first starts with an ambient term, which we model with a three-dimensional vector, as shown:

```
glm::dvec3 color = ka(i) * scene->ambient();
```

Next, for each light, the program computes the direction from the surface point to the light. Using the reflected light vector and shininess, the program computes the specular term. A shadow ray is then constructed from the surface point toward the light. The following code snippet shows our implementation of shadow rays:

```
// Construct shadow ray from surface point toward light
    ray shadowRay(
       P + eps * N,   // origin (offset to avoid self-intersection)
       L,             // direction toward light
       glm::dvec3(1.0),
       ray::SHADOW
    );
```

The function then calls another function, shadowAttenuation(), to check if the light is now blocked. The specific method called depends on the type of light being processed, with directional lights and point lights each having their own shadow attenuation function. The shadowAttenuation function call in our shade function is shown in the code snippet below:

```
// Shadow attenuation
glm::dvec3 shadow = pLight->shadowAttenuation(shadowRay, P);
```

Then the program applies distance attenuation, accumulating diffuse and specular if the object is not in shadow.

cubeMap.cpp

In cubeMap.cpp, the program replaces the background when cube mapping is enabled. In the code, the getColor() function samples the environment cube map when a ray misses all geometry. It does this by first normalizing the ray direction, using the dominant direction component to tell which cube face is hit by the ray (+X, -X, +Y, -Y, +Z, or -Z). The logic to implement this is modeled in the following code snippet:

```
if (ax >= ay && ax >= az) {
    // X face
    if (x > 0) {
        face = 0;  // +X
        u = -z / ax;
        v = -y / ax;
    } else {
        face = 1;  // -X
        u =  z / ax;
        v = -y / ax;
    }
```

Once the coordinates for the corresponding cube face have been calculated, the coordinates are mapped from [-1, 1] to [0,1], clamping the UV coordinates into an acceptable range. This process is shown in the following code snippet:

```
// Clamp for safety
    u = glm::clamp(u, 0.0, 1.0);
    v = glm::clamp(v, 0.0, 1.0);
```

The function getMappedValue() is used to sample the corresponding TextureMap.

trimesh.cpp

The file trimesh.cpp creates the functionality for triangle mesh rendering in our ray tracer. The program first creates meshes by adding vertices for position, normals for smooth shading, colors, and UV coordinates for texture mapping. The program creates a triangle face via the method addFace(). This function also ascertains that the received indices are valid and that the triangle they form has a valid area. The function intersectLocal() checks to see if a ray intersects any triangle in the mesh by iterating over all faces.

This is accomplished by using each triangle's vertices, along with the Möller-Trubore algorithm to see if the ray intersects with the triangle. If the algorithm shows that the triangle is parallel to the ray, intersects with the ray behind the ray itself, or yields an invalid UV coordinate for the intersection point, we know this is not the case; otherwise, we set the barycentric weights used to interpolate normal vectors, UV coordinates, and vertex colors.

If the mesh's normal vectors were set, the vertex normals are barycentrically interpolated for the vertices that make up the current TrimeshFace object's triangle; otherwise, the face

normal is used instead. The following code snippet shows how the TrimeshFace object's vertex normals are interpolated:

```
glm::dvec3 N;
 if (parent->vertNorms && !parent->normals.empty()) {
   const glm::dvec3 &nA = parent->normals[ids[0]];
   const glm::dvec3 &nB = parent->normals[ids[1]];
   const glm::dvec3 &nC = parent->normals[ids[2]];
   N = w * nA + u * nB + v * nC;
 } else {
   N = glm::normalize(glm::cross(e1, e2));
 }
```

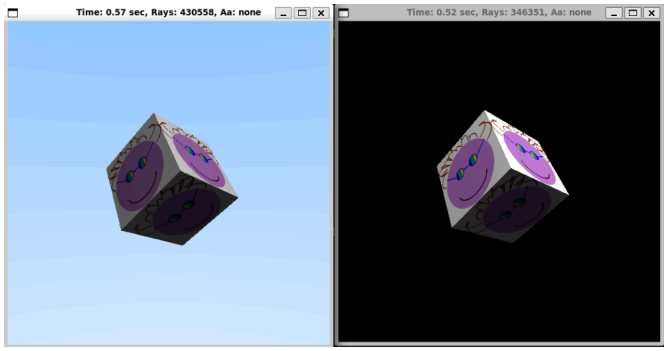
Finally, the color of the intersection point is computed. This is derived in one of three ways: if the parent mesh's UV coordinates are set, the UV coordinates are interpolated for the intersection point to be used for the given texture. If the parent mesh's UV coordinates are set, but its vertex colors are set, then the vertex colors are interpolated for the intersection point instead and used to set the diffuse portion of the point's material. If neither the UV coordinates nor the vertex colors are set, the intersection point's material is set to match the parent mesh's material. The following code snippet shows how the intersection point's material is derived if the parent mesh's vertex colors are set but its UV coordinates are not:

```
else if (!parent->vertColors.empty()) {
   const glm::dvec3 &cA = parent->vertColors[ids[0]];
   const glm::dvec3 &cB = parent->vertColors[ids[1]];
   const glm::dvec3 &cC = parent->vertColors[ids[2]];
```

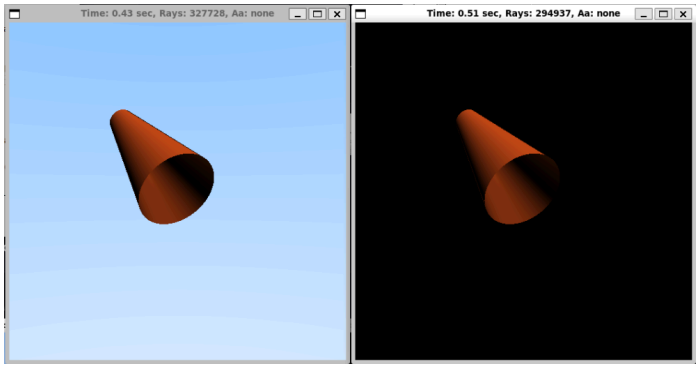
```
    glm::dvec3 c = w * cA + u * cB + v * cC;
    Material m = parent->getMaterial();   // copy the material
    m.setDiffuse(MaterialParameter(c));
        // override diffuse color
    i.setMaterial(m);                    // pass by const reference
  }
```

Results

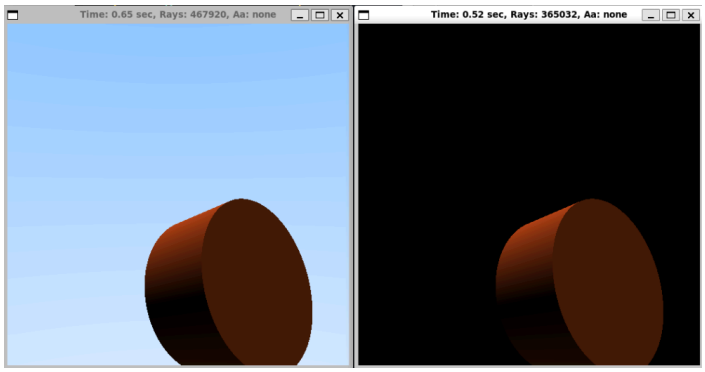
We have a number of photographs comparing our program's rendered images to the example reference images. For this, we used a blue sky background to emphasize the shapes. The following screenshots show various examples of scenes rendered using our ray tracing program:
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name>on the right:



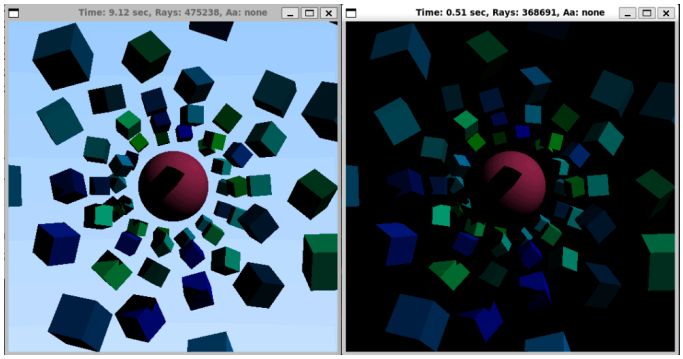
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



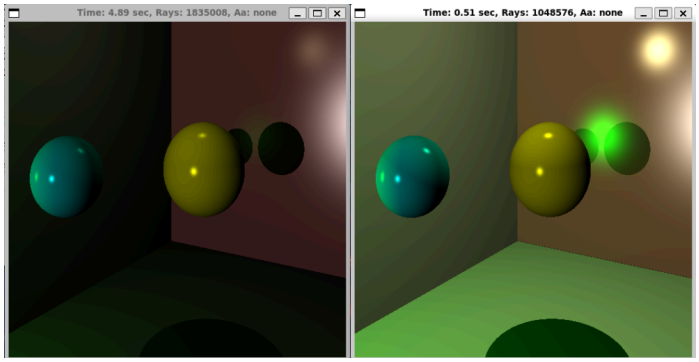
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



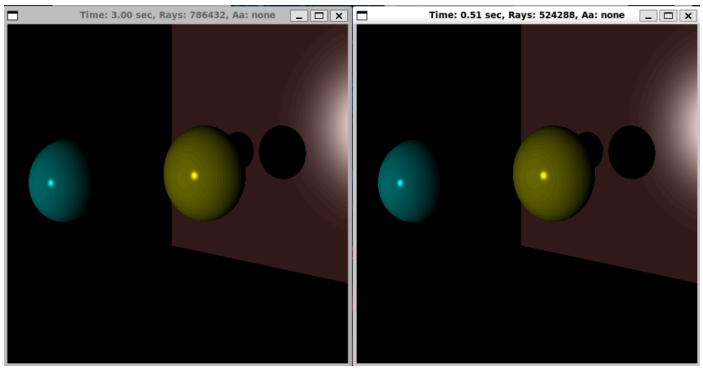
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



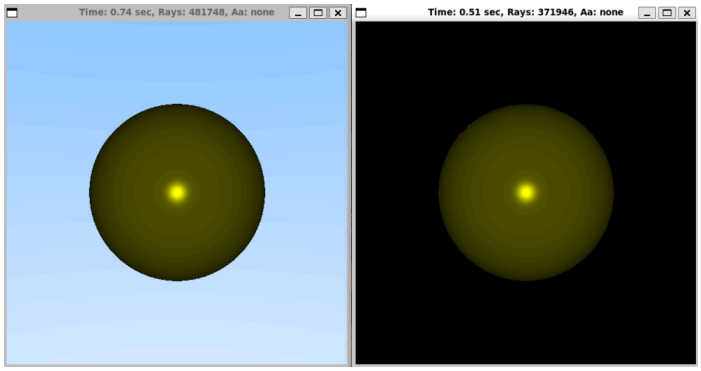
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



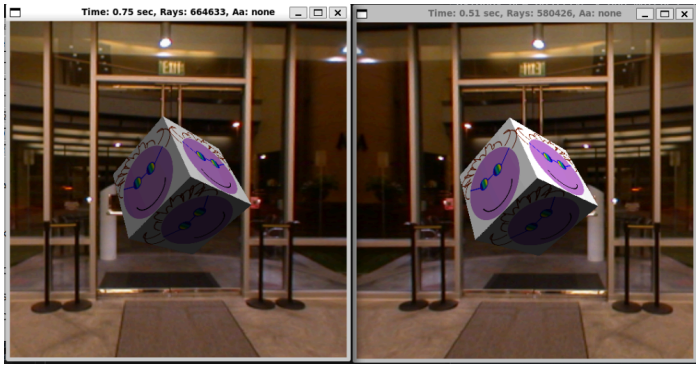
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



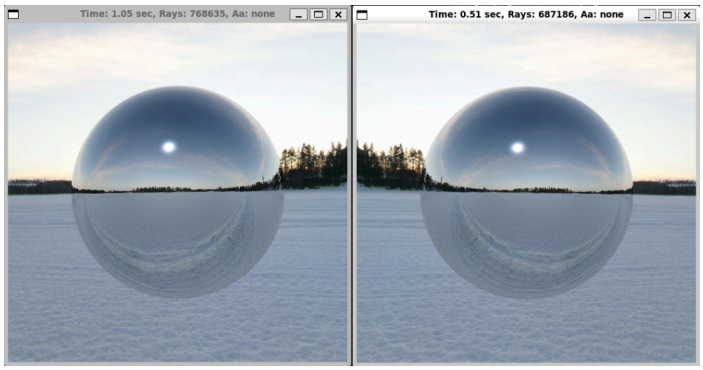
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



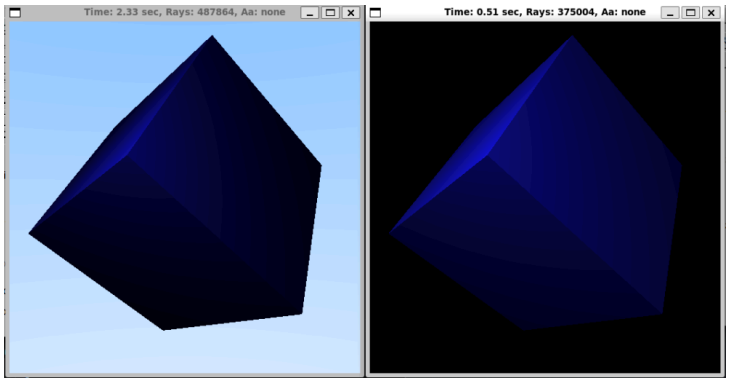
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



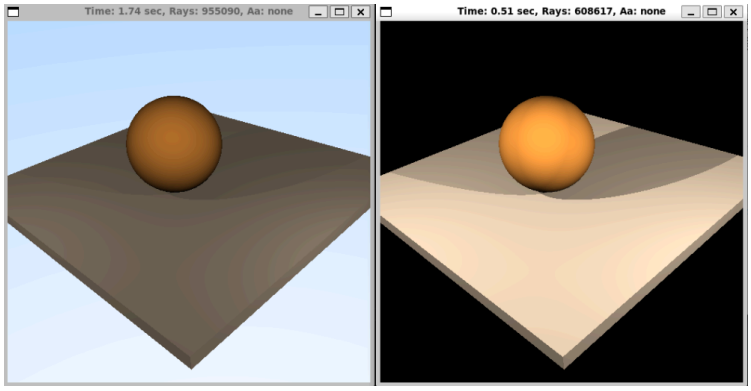
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



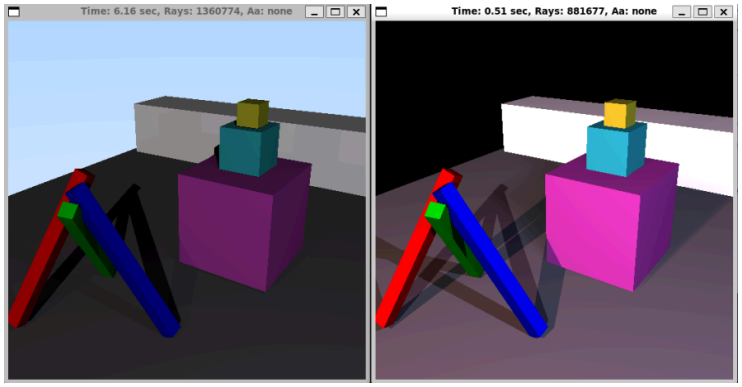
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



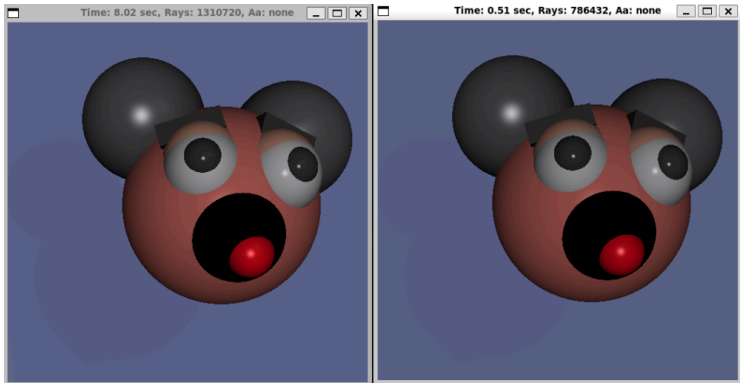
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



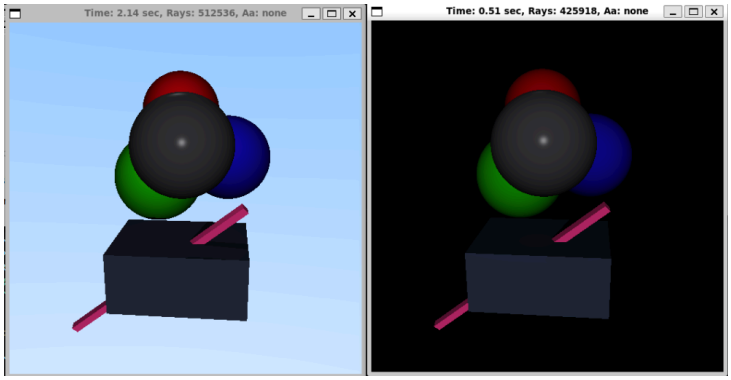
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



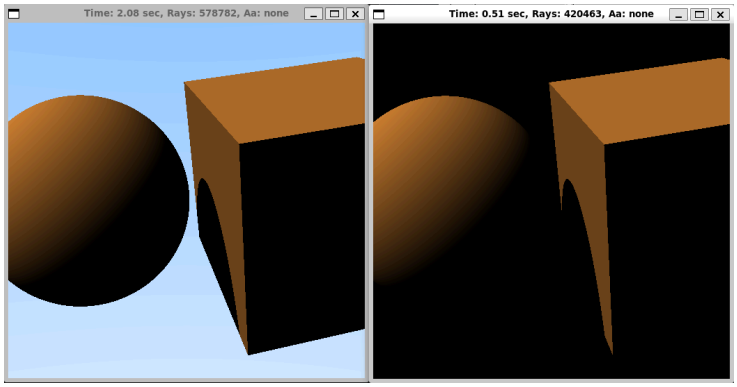
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:

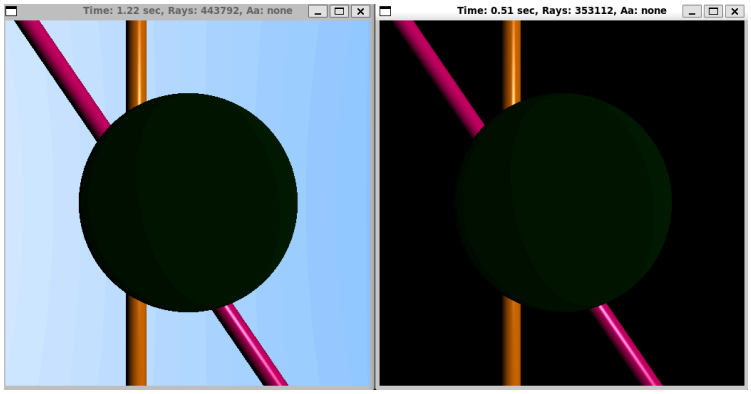


This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:



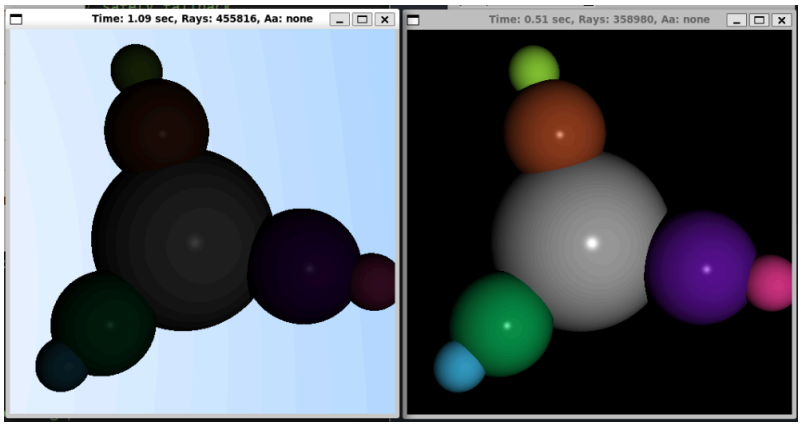
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:

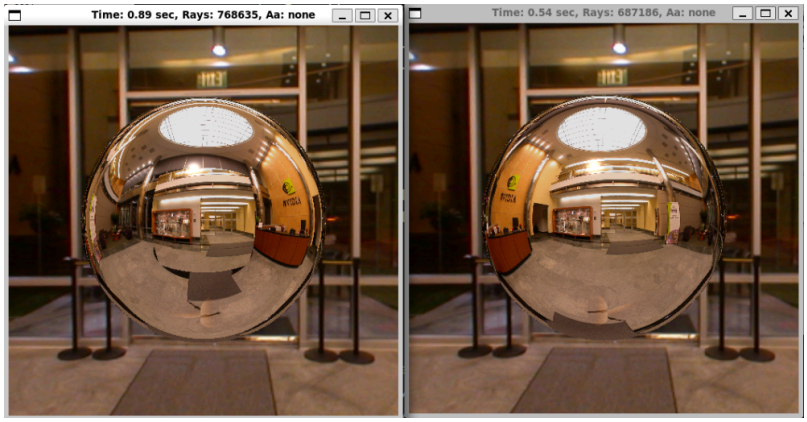
This image shows the comparison between our program's rendered image of <object name> on the left and the reference model of <object name> on the right:

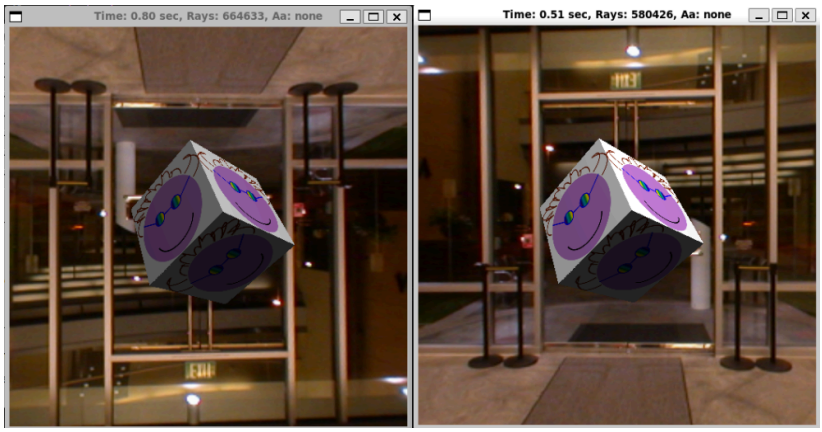


Known Issues and Future Work

Our comparisons of rendered scenes show slight errors in our displays, such as reflections being mirrored, as well as occasional minor lighting differences. Given more time, we would adjust our coordinate and vector calculations to render the reflections more accurately. Here are some examples:







The final image was fixed after changing the return in cubeMapp.cpp for getColor to:

return tMap[face]->getMappedValue(glm::dvec2(u, 1 - v));

Another issue we experienced is that only one of our groupmates was able to successfully load the project solutions, and we had to use his renderings as reference. Given more time, we would ascertain that each team member's libraries and CMake generations are in working order, and that our shared code is executable for each team member's operating system.